Creative Coding with Webcams

Learn how to use live webcam feeds in p5.js sketches.

In p5.js, we're not limited to using pre-recorded videos—we can also incorporate live webcam feeds into our sketches!

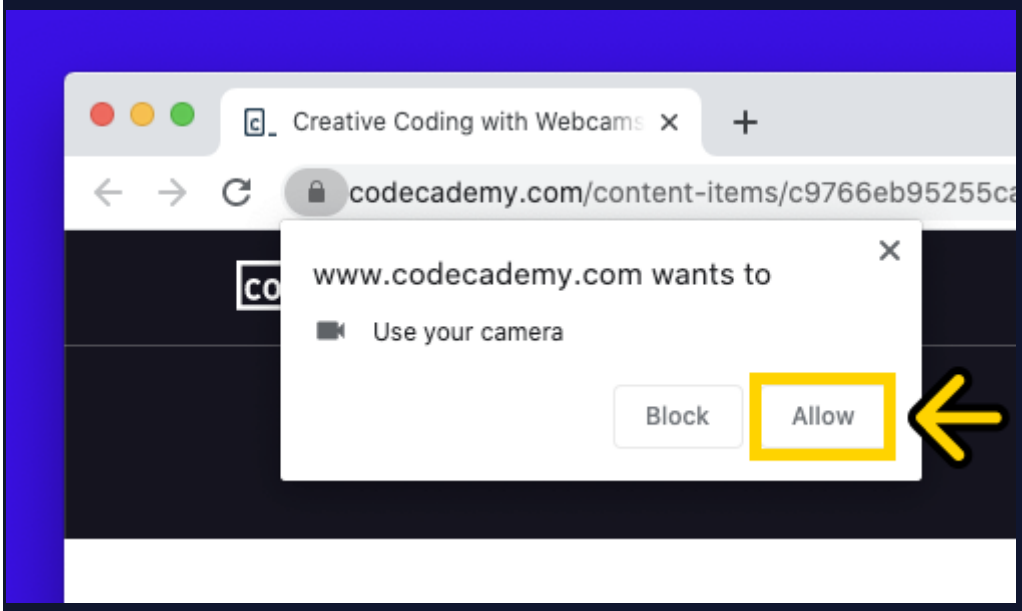
This offers many expressive possibilities; it opens up a whole new dimension of real-time interactivity and can allow you to add your own creative voice into today's technological landscape of live video streaming and face filters.

In this article, you'll:

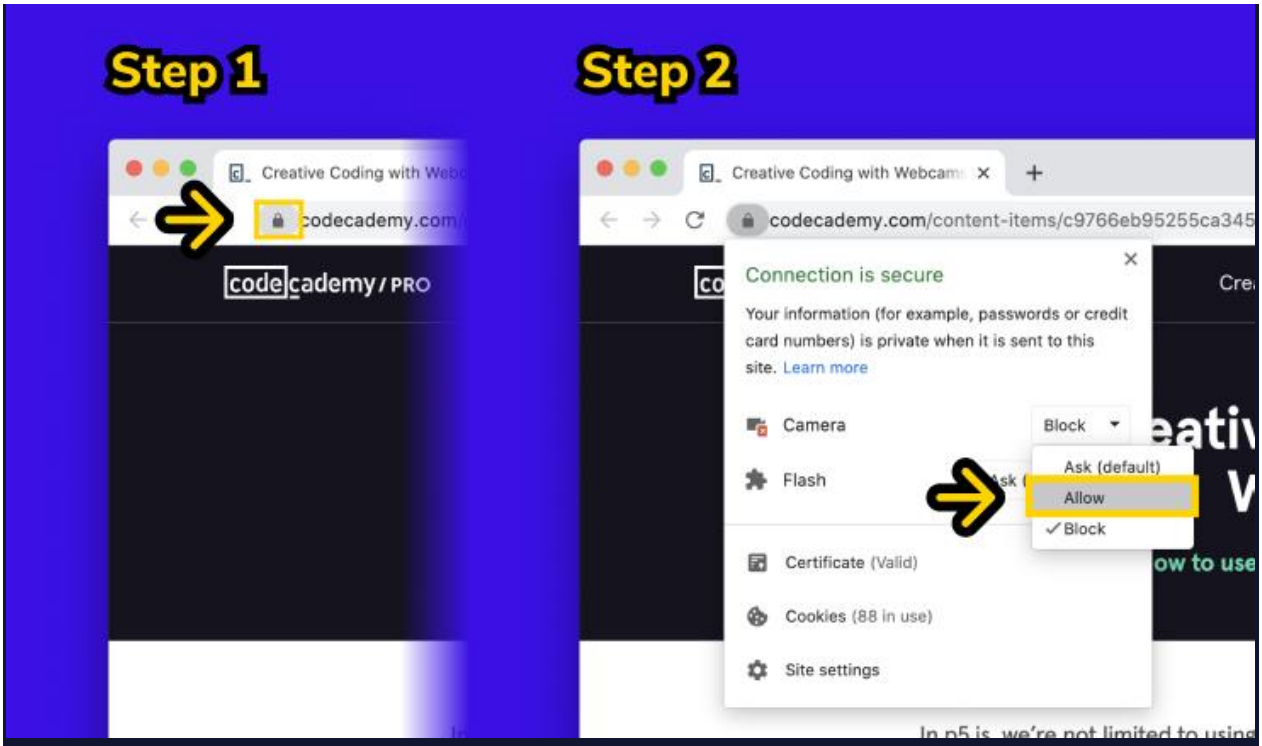
- Learn to incorporate webcam feeds into p5.js sketches.
- Explore examples of creative and interactive applications of webcams in p5.js.

Allowing Your Webcam

To properly view the examples below, you'll need to use a computer with a webcam. When your browser asks to use your camera, make sure to hit the "Allow" button.



Don't worry if you already hit the "Block" button—you can always allow access again by clicking on a lock or camera icon (depending on which web browser you use) next to your browser's address bar. From there, look for your webcam's permission settings and click "Allow".



Let's get started!

Using the Webcam in p5.js

Initializing the Webcam

Incorporating a webcam feed into a p5.js sketch is fairly similar to using a video file. However, one difference is that using the webcam requires a special function: the `createCapture()` function.

The `createCapture()` function can be used to create a webcam capture element in p5.js. As an argument, it requires the type of capture to create—this can be `VIDEO` for capturing the webcam feed or `AUDIO` for the microphone feed. In this article, we'll discuss applications using the `VIDEO` capture element.

```
capture = createCapture(VIDEO); // Creates a webcam capture element
```

When the `createCapture()` function is used to access the webcam, p5.js will create an HTML video element on the webpage that contains the live webcam feed. Just like the HTML video element created when using the `createVideo()` function, the webcam feed's HTML video element sits outside of the canvas.

The `createCapture()` function can take in additional arguments that specify other options: for example, capturing audio alongside video. If you're curious, you can check out the p5.js documentation for the `createCapture()` function.

Drawing the Webcam Capture to the Canvas

In many cases, we'll want to use the webcam feed within the p5.js canvas itself! Just as we do for external videos, we can use the `image()` function for this purpose.

The code below demonstrates how we can use the `image()` function, combined with the `createCapture()` function, to draw the webcam feed to the canvas.

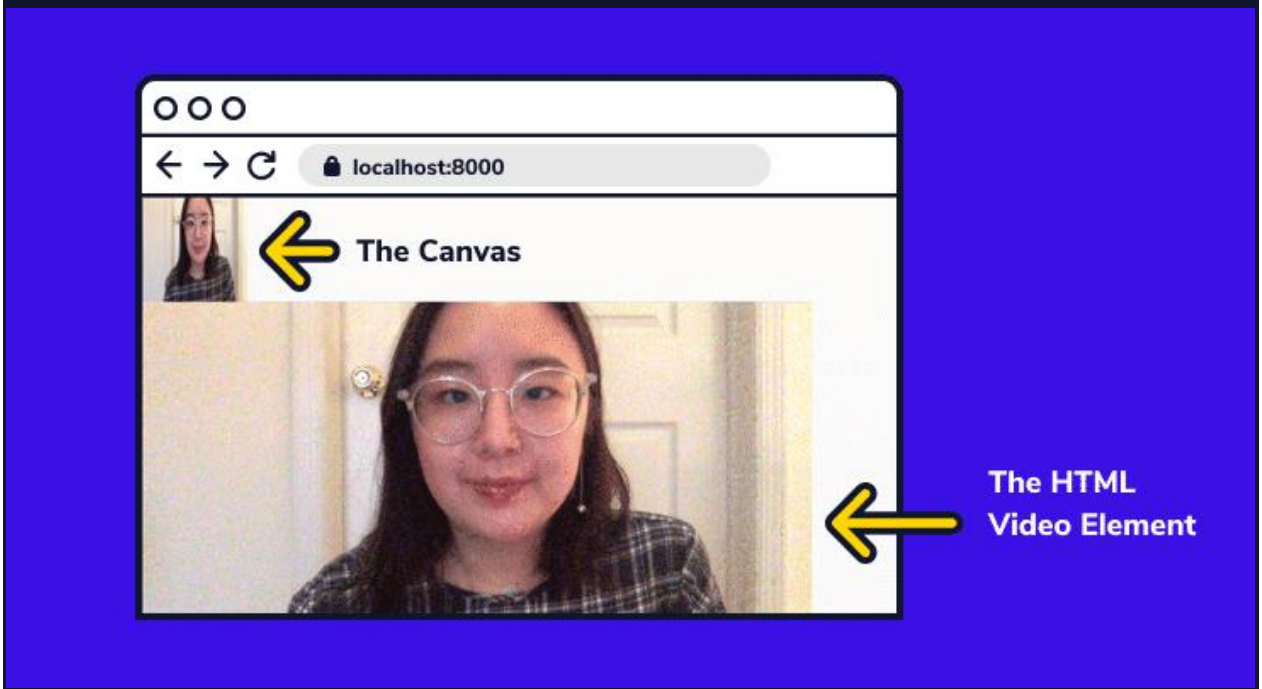
```
let capture;

function setup() {
  createCanvas(200, 200);
  capture = createCapture(VIDEO);  // Creates a webcam capture element
}

function draw() {
  background(220);
  image(capture, 0, 0, width, height); // Draws webcam capture to the canvas
}
```

Above, we use the `createCapture()` function in `setup()` to create a webcam capture element. We then save it to the `capture` variable.

In the `draw()` loop, we call the `image()` function, passing in `capture` to draw the latest webcam frame to the canvas. The last two arguments scale the drawn webcam feed to the canvas' width and height. Here's what this code looks like in action!



If we wanted to get rid of the webcam feed's original HTML video element, we can use the `.hide()` method, as follows.

```
let capture;
function setup() {
  createCanvas(200, 200);
  capture = createCapture(VIDEO);  // Creates a webcam capture element
  capture.hide(); // Hides the original HTML video element
}

function draw() {
  background(220);
  image(capture, 0, 0, width, height); // Draws webcam capture to the canvas
}
```

The code above is the same as the one before, except for one more line added inside the `setup()` function that uses the `.hide()` method. When `.hide()` is called on the webcam capture element, it will remove the HTML video element on the webpage originally created by `createCapture()`—so that all you end up seeing is the webcam feed drawn to the canvas.

Setting the Right Scale

You might have noticed that the scale of the webcam feed in the canvas looks off in the previous example—that's because when we use the `image()` function, we're scaling a normally wide-screen webcam feed to a square canvas.

We can avoid this problem by drawing our webcam on a canvas that's scaled to the dimensions of the webcam feed itself! However, this can be tricky for a few reasons:

- **Not all webcam feed dimensions are the same.** We might be able to create our p5.js sketch knowing the right dimensions of our computer's webcam, but we can't be sure it'll be the case for others who view our sketch using different hardware.
- **The webcam capture takes time to initialize.** It likely won't be properly initialized when the `setup()` function runs (not even when you put it in the `preload()` function!). In other words, we won't be able to access useful information, like the exact pixel dimensions of the video, until later iterations of the `draw()` loop.

One solution is to provide a second argument for the `createCapture()` function to attach a *callback function*. In JavaScript, callback functions are functions that can be passed as an argument to another function. When the first function has finished, the code inside the callback function will run next.

In this case, adding a callback function to `createCapture()` allows us to run any code that needs to execute after the webcam successfully initializes. We can apply this knowledge to solve our scaling problem—after the webcam initializes, we can access the correct width and height of the capture, and use them to scale our canvas accordingly!

For example, to create a canvas that's the full width of the browser window, but scaled to the webcam feed, we can use the following code:

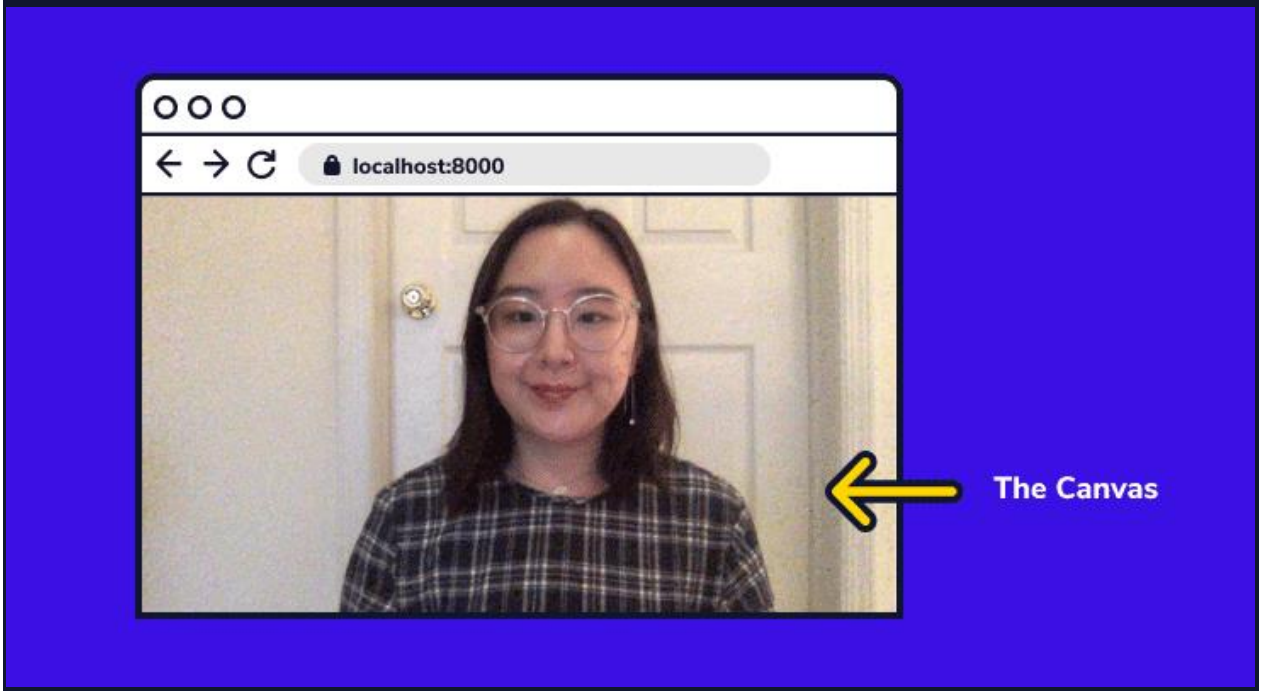
```
let capture;
let desiredWidth, desiredHeight;
let setup(){
  createCanvas(windowWidth, windowHeight);
  capture = createCapture(VIDEO, scaleCanvasToCapture); // Uses a callback function
  capture.hide();
}

function scaleCanvasToCapture() {
  // Sets desired width of canvas to width of the window
  desiredWidth = windowWidth;
  // Calculates height according to webcam feed's scale
  desiredHeight = windowWidth * (capture.height / capture.width);
  // Resizes the canvas to the desired dimensions
  resizeCanvas(desiredWidth, desiredHeight);
  // Uses the .size() method to resize original webcam capture element
  capture.size(desiredWidth, desiredHeight);
}

function draw() {
  image(capture, 0, 0);
}
```

Above, we created the `scaleCanvasToCapture()` function and passed it into `createCapture()` as a callback function. This callback function calculates values for `desiredWidth` and `desiredHeight`, which are the dimensions that we would like our canvas to be resized to.

`desiredWidth` and `desiredHeight` are calculated based off of the webcam's dimensions, which we can now access through the properly-initialized webcam capture element's `.width` and `.height` properties. Finally, we pass these desired dimensions into the `resizeCanvas()` function to scale the canvas accordingly. In the end, our sketch might look something like this:



We'll use this process in all of our interactive examples later on!

Resizing the Webcam Capture Element

The `scaleCanvasToCapture()` function above also contains a new method: the `.size()` method. When applied to a webcam capture element, the `.size()` method resizes it to a specified width and height.

```
capture.size(desiredWidth, desiredHeight);
```

While not absolutely necessary to use the `.size()` method, it can make our lives easier in a few ways:

- When drawing the webcam capture to the canvas later, we don't have to specify arguments for the width and height of the `image()` function—it's already scaled to the right size!
- If we decide to manipulate the webcam capture element's `.pixels` array (which we will later demonstrate), it will be much easier to work from an original capture element that has the same pixel dimensions as the representation of the webcam we draw on the canvas.
- Resizing the capture element smaller can improve performance, especially if you're accessing its `.pixels` array (on the other hand, resizing it too big can potentially create lag problems, which is something you'll need to consider on a case-by-case basis).

Webcams and Security

When using webcams in p5.js (and in web pages in general), you'll need to be mindful of potential challenges due to web security. Loading a webcam feed in a p5.js sketch running locally on your computer may not pose any problems, but you should take several things into account if you host a p5.js webcam sketch on the web.

User Consent

First, remember that viewers need to consent to have their webcam (or audio) used. They'll need to hit the same "Allow" webcam button that should have showed up for you, too, in this article.

This security measure makes a lot of sense—imagine how scary it would be if websites could open your webcam whenever they wanted! Thus, you can't always assume the webcam is accessed and initialized when someone views your p5.js sketch on the web.

HTTPS vs HTTP

Second, you will have to host your p5.js sketch online over a secure protocol, such as HTTPS, as opposed to HTTP. You may recognize these as what goes at the beginning of a URL in your web browser—these are protocols for how a visitor's computer communicates with the website they're currently visiting. HTTPS is more secure because it encrypts the information sent between that viewer and the website's server.

Sometimes (like in the examples in this article), the visitor's webcam data is never actually sent to the website's servers—but that's not the case for many applications, such as video conferencing tools.

To prevent hackers from accessing sensitive video and audio data on the web, many browsers require HTTPS for the webcam and microphone to be accessed. If you're curious about the nitty-gritty security aspects of accessing user media in JavaScript, check out the MDN Documentation.

Creative Webcam Applications

Now that we know the basics of incorporating the webcam, let's get to the fun stuff—exploring interesting and expressive uses of webcams! This is by no means a comprehensive list of what p5.js can do with webcams, but rather a starter kit to help inspire you to think of your own creative applications.

Experimenting with Visuals

Webcam feeds offer the ability to create sketches that experiment with visuals in real-time—we can do this by applying other things we've learned about manipulating images and videos in p5.js!

Photo-booth Filter
One simple way to manipulate the webcam feed is by using the `filter()` function on the canvas. The example below creates a photo-booth effect by applying a filter to the canvas after the webcam capture is drawn.

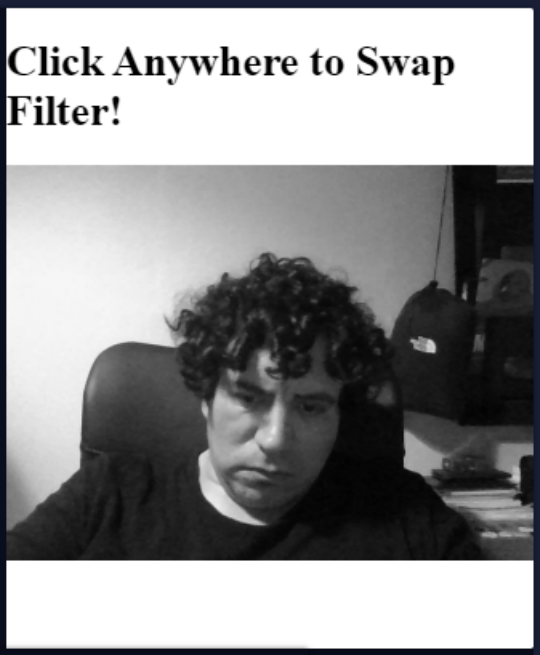
When the mouse is pressed, the sketch cycles through the 3 available filter types stored in `filterTypes` array. We track the current filter type using the `currentFilterIndex` variable to know which type of filter to apply to the canvas in the `draw()` loop. Take a look and play around!

Coding question

Run the sketch below to see the `filter()` function used in combination with the webcam.

Inside the `setup()` function at line 13, take note of how each element in the `filterTypes` array is structured. Try adding to `filterTypes` so the sketch cycles through the `INVERT` filter as well!

```
1   let capture;
2   let desiredWidth, desiredHeight;
3   let currentFilterIndex = 0;
4   let filterTypes;
5
6   function setup() {
7     createCanvas(windowWidth, windowHeight);
8     // Creates webcam capture
9     capture = createCapture(VIDEO,
      scaleCanvasToCapture);
10    capture.hide();
11
12    // TODO: Add an INVERT filter type to
      filterTypes
13    filterTypes = [
14      {name: GRAY},
15      {name: THRESHOLD},
16      {name: POSTERIZE, arg: 3},
17      {name: INVERT}
```



Run

Check answer

You got it!

We can also apply our knowledge of pixel manipulation to webcam feeds! Just like image and video elements, webcam capture elements have their own `.pixels` array. After loading this array, you can access and manipulate individual pixels within each frame of the webcam feed.

Rainbow Heart Emoji Mirror

Sometimes, the most interesting uses of webcam don't even involve drawing the webcam feed to the canvas itself. The example below features an "emoji mirror" filled with rainbow hearts. It's created by loading the `.pixels` array of the webcam capture and accessing their RGBA values—but it does not manipulate the pixels, nor draw the webcam feed itself to the canvas.

Before we load the pixels and do anything using the webcam feed in the `draw()` loop, the `isWebcamReady` variable is used to check if the webcam properly initialized. In the

callback function of `createCapture()`, we set `isWebcamReady` to `true`—in addition to calling our `scaleCanvasToCapture()` function. This step isn't absolutely necessary for the sketch to work, but it can be useful if we want the code in the `draw()` loop to only run when there's a properly initialized webcam element! We'll use `isWebcamReady` in some following sketches, where it's helpful to wait until the webcam's ready.

After loading the `.pixels` array, the sketch iterates over every 8th pixel using the `stepSize` variable in the nested `for` loop. Then, the RGBA values for each pixel are used to calculate a brightness value using a special p5.js function—the `brightness()` function. This brightness value is later used in a series of `if` statements to draw an emoji heart to the canvas, using the `text()` function.

Run the code to see yourself on the canvas as a rainbow of heart emojis!


```
let capture;
let desiredWidth, desiredHeight;
let isWebcamReady = false;
// TODO: Change stepSize to a different number
```

```
let stepSize = 8;

function preload() {
  createCanvas(windowWidth, windowHeight);
  // Create webcam capture
  capture = createCapture(VIDEO, function(){
    scaleCanvasToCapture();
    isWebcamReady = true;
  });
  capture.hide();
}

function draw() {
  background(255);
  if (isWebcamReady){
    capture.loadPixels();
    // Iterates through pixels in capture, in intervals of 8
    for (let y = 0; y < capture.height; y += stepSize) {
      for (let x = 0; x < capture.width; x += stepSize) {
        let indexOfRed = (x + y * capture.width) * 4;
        // Gets color of current pixel
        let r = capture.pixels[indexOfRed];
        let g = capture.pixels[indexOfRed + 1];
        let b = capture.pixels[indexOfRed + 2];
        let a = capture.pixels[indexOfRed + 3];
        // Gets the brightness of the pixel
        let brightness_val = brightness([r, g, b, a]);
        textSize(stepSize - 2);
        // Draws an emoji here given the brightness value
        if (brightness_val > 80){
          text('♡', x, y);
        } else if (brightness_val > 70){
          text('🤍', x, y);
        } else if (brightness_val > 60){
          text('🩶', x, y);
        } else if (brightness_val > 50) {
          text('♡', x, y);
        } else if (brightness_val > 40) {
          text('💔', x, y);
        } else if (brightness_val > 30) {
```

```
          text('🤍', x, y);
        } else if (brightness_val > 20) {
          text('🤍', x, y);
        } else {
          text('🤍', x, y);
        }
      }
    }
  }
}

function scaleCanvasToCapture() {
  // Sets desired width of canvas to width of the window
  desiredWidth = windowWidth;
  // Calculates height according to webcam feed's scale
  desiredHeight = windowWidth * (capture.height / capture.width);
  // Resizes the canvas to the desired dimensions
  resizeCanvas(desiredWidth, desiredHeight);
  // Uses the .size() method to resize original webcam capture element
  capture.size(desiredWidth, desiredHeight);
}
```

Experimenting with Time

Another exciting application of webcams is creating sketches that experiment with time. Using what we know about creating animations in p5.js, we can leverage the `draw()` loop to combine webcam feed frames in visually interesting ways.

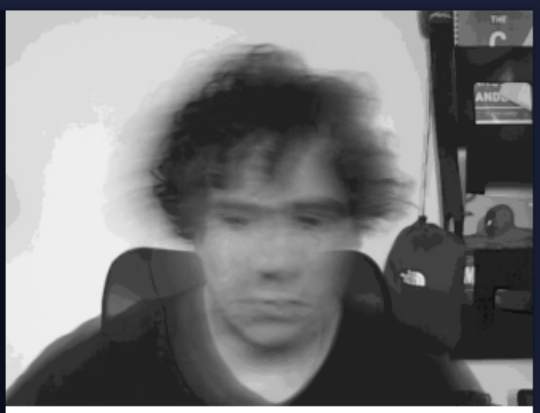
Ghost-like Motion Blur
Instead of only drawing the current frame, we can, for example, layer video frames on top of each other. The example below creates a ghost-like effect by making each webcam frame semi-transparent. This is done by loading the webcam's `.pixels` array and manipulating each pixel's alpha value. The `background()` function is never called, so the video frames layer on top of each other to create a blurred effect. Finally, the `filter()` function applies a grayscale effect over the entire canvas.

Coding question

Run the sketch below, and move around in front of the camera to see the ghost-like effect in action.

At line 4, what happens if you set the `alphaVal` variable to other values between `0` and `255`?

```
1   let capture;
2   let desiredWidth, desiredHeight;
3   // TODO: Change alphaVal to a different
    number between 0 and 255
4   let alphaVal = 15;
5
6   function setup() {
7     createCanvas(windowWidth, windowHeight);
8     capture = createCapture(VIDEO, function
      () {
9       scaleCanvasToCapture();
10      isWebcamReady = true;
11    });
12    capture.hide();
13  }
14
15  function draw() {
16    // Loads pixels of capture
17    capture.loadPixels();
```



Run

Check answer

Time Warp Scan Effect

With a little bit more elbow grease, we can create more complex effects! The following sketch uses a combination of techniques in the p5.js course to recreate the "time warp scan" (or slit-scan) effect currently popular on many social media apps. The effect features a horizontal, laser-like line that "scans" the webcam feed from top to bottom, freezing the pixels that it crosses to create a warped effect.

To simulate the laser-like line, we can use the `line()` function with a counter that increments its position in the y-axis. As it's crossing the canvas, we'll use the `.get()` method on the webcam capture element to access the short, rectangular region that the laser line currently crosses as an image element and draw it on the canvas directly above the laser line. Because there's no call to the `background()` function in the `draw()` loop, these rectangular regions will stack together to create the ongoing, warped image.

To draw the live, non-warped part of the webcam feed, we use the `image()` function to draw the latest frame of the webcam to the canvas, positioning it right below the laser

line. In the code, we supply the `image()` function with an additional 4 optional arguments (for a total of 9 arguments) that draw a cropped, rectangular region of the webcam frame to the canvas, as follows:

```
image(sourceElement, dx, dy, dWidth, dHeight, sx, sy, sWidth, sHeight);
```

You may be familiar with the first 5 arguments that start with "d" (for destination): the source image or video element, the (x, y) position, and a width and height for scale. The next 4 arguments—starting with "s" (for source)—stand for the (x, y) position, the width, and the height that define the rectangular region of the source image or video frame that we want to draw. To get a better picture of how these optional arguments work, view a helpful diagram on p5.js' documentation of the `image()` function.

For this time warp scan effect, we supplied the `image()` function with these optional arguments to draw the part of the webcam feed that should appear below the moving laser line—calculating these arguments based off of the canvas' dimensions and the current laser line position. Check it out below!

Coding question

Run the sketch to view the time warp scan effect in action!

At line 7, try experimenting with different values of the `fps` variable, which stands for "Frames per Second" and is passed into the `frameRate()` function at the bottom of the `draw()` loop. How does the time warp scan effect change as `fps` changes?

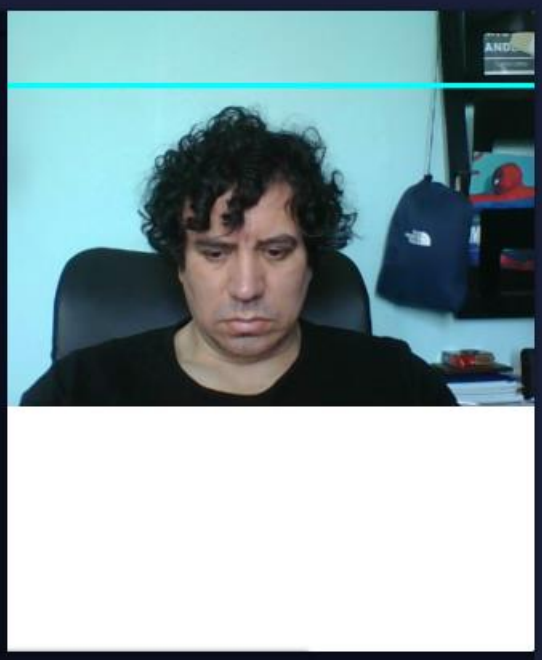
```
    laserPositionY);

  // Resets laser line to the top
  if (laserPositionY > desiredHeight){
    laserPositionY = 0;
  }

  // Increments laser line position
  laserPositionY++;

  // Adjusts frame rate to slow down
  frameRate(fps);
 }
}

function scaleCanvasToCapture() {
  // Sets desired width of canvas to
  width of the window
  desiredWidth = windowWidth;
```



Run

Check answer

Advanced Experiment: Pose Recognition

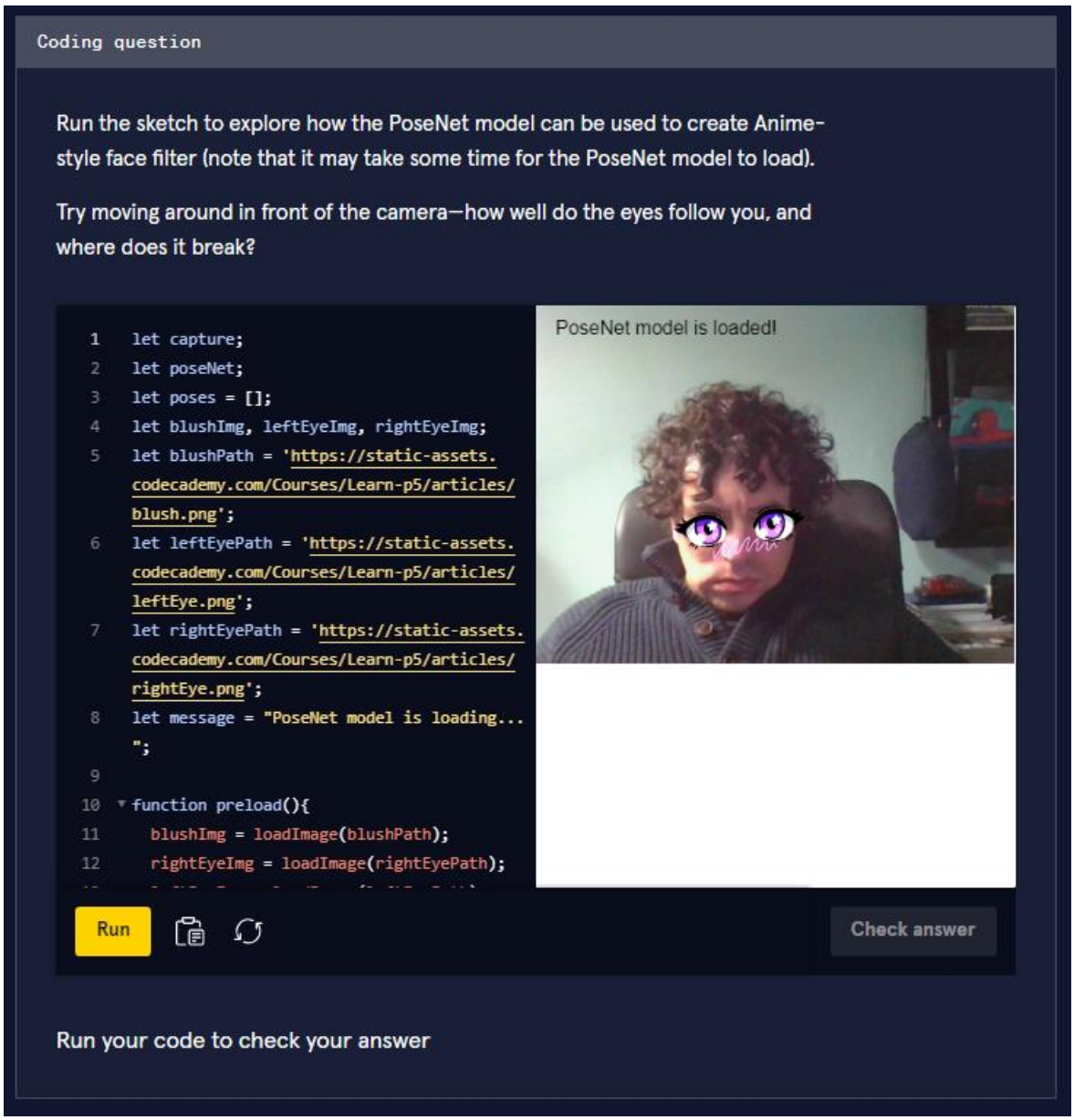
Arguably, some of the most fun and interactive uses of the webcam are sketches that respond to input like poses, gestures, and facial expressions. Doing this, however, will often require additional JavaScript libraries, outside of p5.js, that can convert a video feed into pose data. If you're comfortable with p5.js and JavaScript, and excited to explore different tools and techniques, this section is for you!

Luckily, some resources are designed to integrate smoothly with p5.js. One such resource is ml5.js's *PoseNet*. ml5.js is an open-source library for creating and exploring machine learning in browser-based applications, and PoseNet is a machine learning model for "Real-time Human Pose Estimation". PoseNet works by using trained data to detect body parts in an image or video—you can read more about how it works in this Medium article.

Anime-style Face Filter

The sketch below uses PoseNet to create an Anime-style face filter. After detecting a human pose in the webcam feed, PoseNet returns data about the body parts it detects, giving the predicted x and y locations of each body part in the webcam frame. In this sketch, we specifically access the positions of the right eye, left eye, and nose and use them to calculate where to draw (and what angle to rotate) images for cartoon eyes and blush.

This code introduces many new concepts and functions from the ml5.js library, so don't worry about understanding every part, line-by-line (unless you'd like to dig deeper on your own)! Consider this example as a taste of what's possible by combining ml5.js and p5.js, and a springboard for further exploration.



Additional Considerations

Joining webcam feeds with image recognition can not only be fun, but also help you gain literacy about the technological systems behind the scenes!

While the example above is fun and playful, there are many considerations to take into account if applying body or facial recognition in day-to-day technology—such as making sure a diversity of faces and bodies can be recognized. On the other hand, it's important to be aware of the ways these systems can also be harmful—when they are used without consent, encode bias, or used to treat people unjustly.

That being said, making creative experiments with tools like ml5.js and p5.js can be a fun and engaging start to familiarize yourself with how these systems work and help you have the language to shape technology for the better.

Experiment Yourself!

This article should provide you with a foundation for how to integrate webcam feeds into p5.js sketches, as well as offer a taste of creative, expressive, and playful possibilities.

By exploring some of the technical processes behind many popular camera-based applications, you have a starting point for how to understand (and creatively shape!) today's technological landscape of smartphones, live video streaming, and image recognition.

As you create your own p5.js sketches, either locally on your computer or in the p5.js web editor, what new uses can you imagine for the webcam? How can you create moments that foster play, inspire artistic expression, or improve daily life?